

# StVEC: A Vector Instruction Extension for High Performance Stencil Computation

**Naser Sedaghati**   Renji Thomas   Louis-Noël Pouchet  
Radu Teodorescu   P. Sadayappan

Department of Computer Science and Engineering  
The Ohio State University

HPC Research Lab: [barista.cse.ohio-state.edu](http://barista.cse.ohio-state.edu)

Computer Architecture Lab: [arch.cse.ohio-state.edu](http://arch.cse.ohio-state.edu)

October 13<sup>th</sup> 2011

- 1 Introduction
- 2 Vectorization of Stencils
- 3 Enhancing Vector ISA with StVEC
- 4 Generating Code for StVEC
- 5 Evaluation
- 6 Summary

# Stencil Computation

## Repeat over TIME

- Sweep over a spatial grid
- Compute a point from neighbor points values
  - Same grid or multiple grids

## Numerous application domains

- Finite difference methods for solving PDEs
- Image processing (e.g. MRI image pipeline)
- Computational electromagnetics, CFD, numerical relativity, etc.

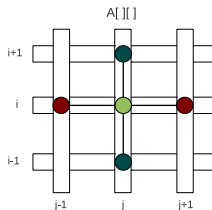
# Stencil Computation: An Example

## 2-D 5-point Jacobi

```

for (t = 0; t < TMAX; t++)
  for (i = 1; i < N - 1; i++)
    for (j = 1; j < M - 1; j++)
      B[i][j] =
        A[i-1][j] +
        A[i][j-1] + A[i][j] + A[i][j+1] +
        A[i+1][j];

```



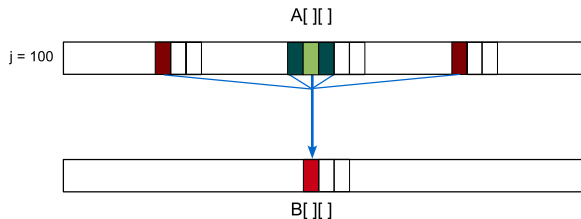
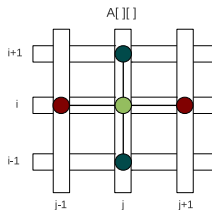
# Stencil Computation: An Example

## 2-D 5-point Jacobi

```

for (t = 0; t < TMAX; t++)
  for (i = 1; i < N - 1; i++)
    for (j = 1; j < M - 1; j++)
      B[i][j] =
        A[i-1][j] +
        A[i][j-1] + A[i][j] + A[i][j+1] +
        A[i+1][j];

```



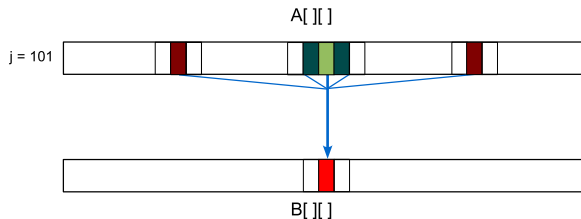
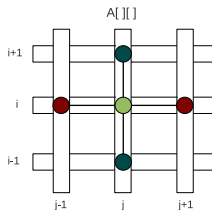
# Stencil Computation: An Example

## 2-D 5-point Jacobi

```

for (t = 0; t < TMAX; t++)
  for (i = 1; i < N - 1; i++)
    for (j = 1; j < M - 1; j++)
      B[i][j] =
        A[i-1][j] +
        A[i][j-1] + A[i][j] + A[i][j+1] +
        A[i+1][j];

```

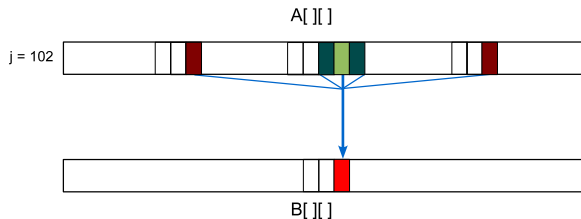
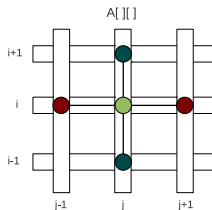


# Stencil Computation: An Example

## 2-D 5-point Jacobi

```

for (t = 0; t < TMAX; t++)
  for (i = 1; i < N - 1; i++)
    for (j = 1; j < M - 1; j++)
      B[i][j] =
        A[i-1][j] +
        A[i][j-1] + A[i][j] + A[i][j+1] +
        A[i+1][j];
  
```



# Short-Vector SIMD

## Identical computation on small chunks of data

- Independent operations
- Vector size (width) of 2 to 64
- Packing operations to form a vector (shuffle, extract, etc.)

## SIMD performance

- Multiple SIMD units per CPU
- Maximum speedup equals the vector width

## Ubiquitous features on modern processors

- x86 – SSE, AVX
- Power – VMX/VSX
- ARM – NEON
- Cell SPU

# Vectorization: An Example

Vector width = 4, N divisible by 4

```
for (t = 0; t < T; t++)  
  for (i = 4; i < N; i++)  
    A[i] = B[i] * B[i] ;
```

# Vectorization: An Example

Vector width = 4, N divisible by 4

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] = B[i] * B[i] ;
```

1: ASM (MIPS-like)

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++){
    LD  R1, &B[i]
    MUL R2, R1, R1
    ST  R2, &A[i]
  }
```

# Vectorization: An Example

Vector width = 4, N divisible by 4

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] = B[i] * B[i] ;
```

1: ASM (MIPS-like)

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++){
    LD  R1, &B[i]
    MUL R2, R1, R1
    ST  R2, &A[i]
  }
```

2: 4-way unroll + re-schedule

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i+=4){
    LD  R1, &B[i]
    LD  R2, &B[i+1]
    LD  R3, &B[i+2]
    LD  R4, &B[i+3]
    MUL R5, R1, R1
    MUL R6, R2, R2
    MUL R7, R3, R3
    MUL R8, R4, R4
    ST  R5, &A[i]
    ST  R6, &A[i+1]
    ST  R7, &A[i+2]
    ST  R8, &A[i+3]
  }
```

# Vectorization: An Example

Vector width = 4, N divisible by 4

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] = B[i] * B[i] ;
```

1: ASM (MIPS-like)

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++){
    LD  R1, &B[i]
    MUL R2, R1, R1
    ST  R2, &A[i]
  }
```

2: 4-way unroll + re-schedule

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i+=4){
    LD  R1, &B[i]
    LD  R2, &B[i+1]
    LD  R3, &B[i+2]
    LD  R4, &B[i+3]
    MUL R5, R1, R1
    MUL R6, R2, R2
    MUL R7, R3, R3
    MUL R8, R4, R4
    ST  R5, &A[i]
    ST  R6, &A[i+1]
    ST  R7, &A[i+2]
    ST  R8, &A[i+3]
  }
```

3: Vectorize

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i+=4){
    VLD VR1, &B[i]
    VMUL VR2, VR1, VR1
    VST  VR2, &A[i]
  }
```

# Vectorization: An Example

Vector width = 4, N divisible by 4

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] = B[i] * B[i] ;
```

1: ASM (MIPS-like)

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++){
    LD  R1, &B[i]
    MUL R2, R1, R1
    ST  R2, &A[i]
  }
```

2: 4-way unroll + re-schedule

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i+=4){
    LD  R1, &B[i]
    LD  R2, &B[i+1]
    LD  R3, &B[i+2]
    LD  R4, &B[i+3]
    MUL R5, R1, R1
    MUL R6, R2, R2
    MUL R7, R3, R3
    MUL R8, R4, R4
    ST  R5, &A[i]
    ST  R6, &A[i+1]
    ST  R7, &A[i+2]
    ST  R8, &A[i+3]
  }
```

3: Vectorize

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i+=4){
    VLD VR1, &B[i]
    VMUL VR2, VR1, VR1
    VST  VR2, &A[i]
  }
```

## Observation

- Aligned memory referencing (i.e. B[i]) helps vectorization!

## Vectorization of Stencils

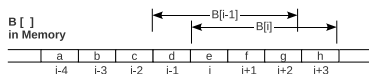
# Vectorizing Stencil Computation

```

for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] += B[i-1] * B[i];

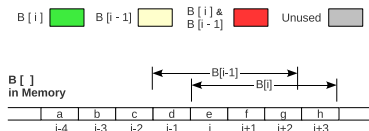
```

$B[i]$  
 $B[i-1]$  
 $B[i] \&$   
 $B[i-1]$  
 Unused 



# Vectorizing Stencil Computation

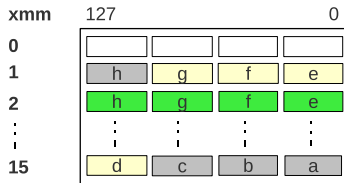
```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] += B[i-1] * B[i];
```



**Solution1:** load + shuffle

$B[]$  in XMM Registers

SSE Assembly ( $N=1024$ )

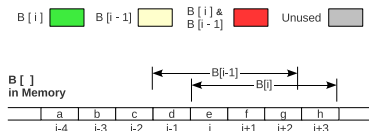


```

LOOP:
movaps    16+B(...), %xmm2
movaps    %xmm2, %xmm1
palignr   $12, B(...), %xmm1
mulps    %xmm2, %xmm1
addps    16+A(...), %xmm1
movaps    %xmm1, 16+A(...)
addq     $4, %rdx
cmpq     $1020, %rdx
jnb      LOOP
  
```

# Vectorizing Stencil Computation

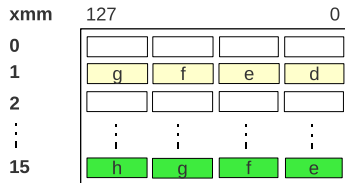
```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] += B[i-1] * B[i];
```



Solution1: load + shuffle  
 Solution2: unaligned load

$B[]$  in XMM Registers

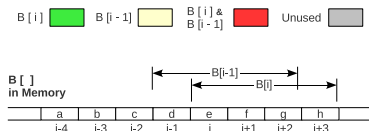
SSE Assembly (N=1024)



```
LOOP:
movups    12+B(...), %xmm1
mulps    16+B(...), %xmm1
addps    16+A(...), %xmm1
movaps    %xmm1, 16+A(...)
addq     $4, %rdx
cmpq     $1020, %rdx
jnb     LOOP
```

# Vectorizing Stencil Computation

```
for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] += B[i-1] * B[i];
```



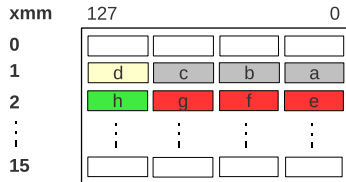
**Solution1:** load + shuffle

**Solution2:** unaligned load

**Our Solution:** StVEC (no shuffle, no unaligned load)

$B[]$  in XMM Registers

SSE Assembly (N=1024)



```
LOOP:
movaps    16+B(...), %xmm2
stmulps  $3, %xmm1, %xmm2, %xmm2
addps    16+A(...), %xmm2
movaps    %xmm2, 16+A(...)
; circulate the buffer(s)
addq     $4, %rdx
cmpq     $1020, %rdx
jb       LOOP
```

## Enhancing Vector ISA with StVEC

# Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils

# Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

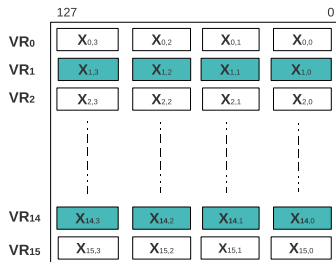
- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e.  $VOPR_x$ ) with two source regs
  - *base* and *extension*

# Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e.  $VOPR_x$ ) with two source regs
  - *base* and *extension*

16x128-bit vector register file



base =  $VR_1$  , extension =  $VR_{14}$

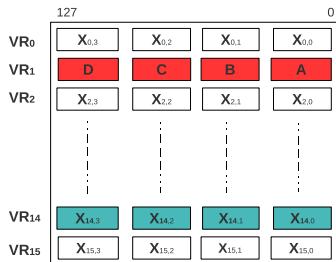


# Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e.  $VOPR_x$ ) with two source regs
  - base* and *extension*

16x128-bit vector register file



base =  $VR_1$  , extension =  $VR_{14}$

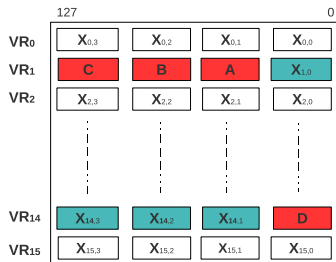
| source | offset | $VOPR_x$              |
|--------|--------|-----------------------|
| $VR_1$ | 0      | $X_{1,0:4}$ (aligned) |

# Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e.  $VOPR_x$ ) with two source regs
  - base* and *extension*

16x128-bit vector register file



base =  $VR_1$  , extension =  $VR_{14}$

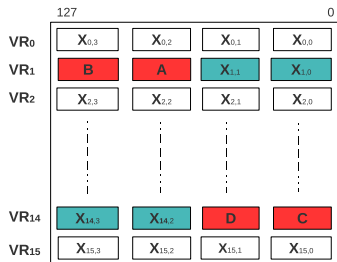
| source          | offset | $VOPR_x$              |
|-----------------|--------|-----------------------|
| $VR_1$          | 0      | $X_{1,0:4}$ (aligned) |
| $VR_1, VR_{14}$ | 1      | $X_{1,1:3}X_{14,0:1}$ |

# Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e.  $VOPR_x$ ) with two source regs
  - base* and *extension*

16x128-bit vector register file



base =  $VR_1$  , extension =  $VR_{14}$

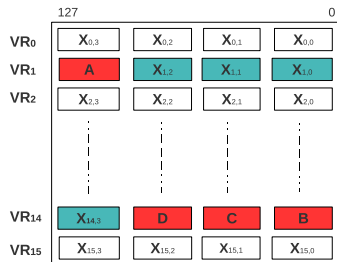
| source          | offset | $VOPR_x$              |
|-----------------|--------|-----------------------|
| $VR_1$          | 0      | $X_{1,0:4}$ (aligned) |
| $VR_1, VR_{14}$ | 1      | $X_{1,1:3}X_{14,0:1}$ |
| $VR_1, VR_{14}$ | 2      | $X_{1,2:2}X_{14,0:2}$ |

# Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e.  $VOPR_x$ ) with two source regs
  - base* and *extension*

16x128-bit vector register file



base =  $VR_1$  , extension =  $VR_{14}$

| source          | offset | $VOPR_x$               |
|-----------------|--------|------------------------|
| $VR_1$          | 0      | $X_{1,0:4}$ (aligned)  |
| $VR_1, VR_{14}$ | 1      | $X_{1,1:3} X_{14,0:1}$ |
| $VR_1, VR_{14}$ | 2      | $X_{1,2:2} X_{14,0:2}$ |
| $VR_1, VR_{14}$ | 3      | $X_{1,3:1} X_{14,0:3}$ |

# StVEC Instructions

## StVEC operands

Target: register-register vector instructions

- **src1** and **dst**: unchanged
- **src2**: expanded to: **offset**, **base** and **extension**

# StVEC Instructions

## StVEC operands

Target: register-register vector instructions

- **src1** and **dst**: unchanged
- **src2**: expanded to: **offset**, **base** and **extension**

SSE translation to StVEC (vector width:  $W = 4$ )

### SSE

- *mulps*  $VR_x, VR_y$

### StVEC

- *stmulps* **offset**,  $VR_x, VR_z, VR_y$

# StVEC Instructions

## StVEC operands

Target: register-register vector instructions

- **src1** and **dst**: unchanged
- **src2**: expanded to: **offset**, **base** and **extension**

## SSE translation to StVEC (vector width: $W = 4$ )

### SSE

- *mulps*  $VR_x, VR_y$
- $VR_y = VR_x * VR_y$

### StVEC

- *stmulps* *offset*,  $VR_x$ ,  $VR_z$ ,  $VR_y$
- $VR_y = VR_x\{offset : W - offset\} VR_z\{0 : offset\} * VR_y$

```

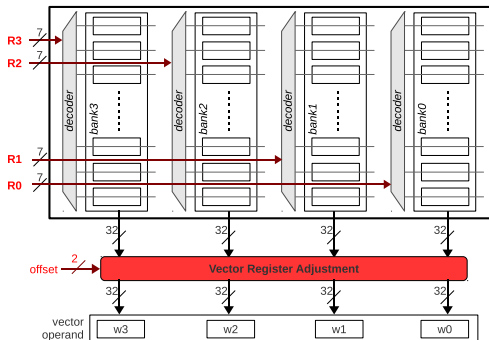
LOOP:
movaps    16+B(...), %xmm2
stmulps  $3, %xmm1, %xmm2, %xmm2
addps    16+A(...), %xmm2
movaps    %xmm2, 16+A(...)
; circulate the buffer(s)
addq     $4, %rdx
cmpq     $1020, %rdx
jb       LOOP
  
```

# Modified Vector Register File (StVRF)

## Modifications to the baseline VRF (BVRF)

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

$$T_{StVRF} \approx T_{BVRF} + T_{VRA}$$



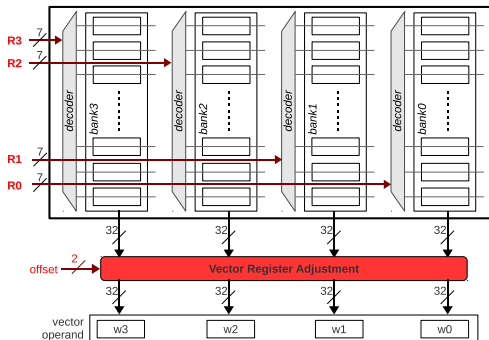
# Modified Vector Register File (StVRF)

## Modifications to the baseline VRF (BVRF)

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

Example: `stmulps $1, %xmm1, %xmm2, %xmm3`

- vector width = 4
- offset = 1
- base = xmm1
- extension = xmm2
- src1 = dst = xmm3
- src2 = `xmm1{1:3}xmm2{0:1}`
- OP = vector multiply





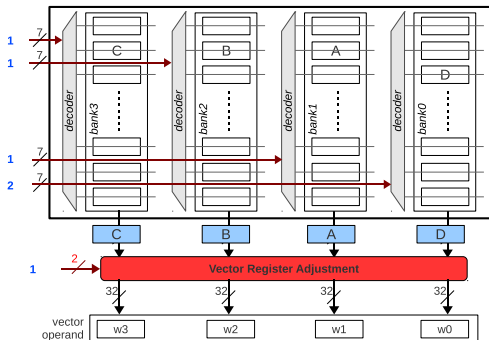
# Modified Vector Register File (StVRF)

## Modifications to the baseline VRF (BVRF)

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

Example: `stmulps $1, %xmm1, %xmm2, %xmm3`

- vector width = 4
- offset = 1
- base = xmm1
- extension = xmm2
- src1 = dst = xmm3
- src2 = `xmm1{1:3}xmm2{0:1}`
- OP = vector multiply



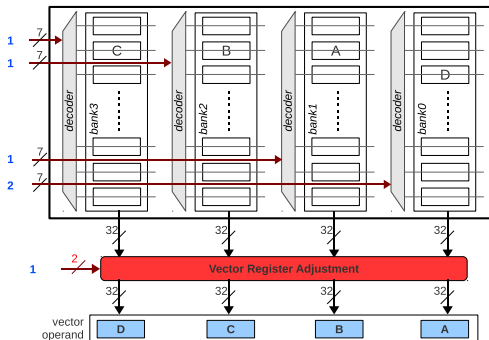
# Modified Vector Register File (StVRF)

## Modifications to the baseline VRF (BVRF)

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

Example: `stmulps $1, %xmm1, %xmm2, %xmm3`

- vector width = 4
- offset = 1
- base = xmm1
- extension = xmm2
- src1 = dst = xmm3
- src2 = `xmm1{1:3}xmm2{0:1}`
- OP = vector multiply



## Generating Code for StVEC

# The Code Generation Procedure

**Input:** Abstract syntax tree (AST) of a vectorizable innermost loop

- 1 generate basic intrinsics
- 2 perform StVEC code generation

**Output:** vectorized loop with StVEC intrinsics

# StVEC Code Generation

**Input:** basic intrinsics loop

## The proposed algorithm

- 1 replace every unaligned reference by two aligned loads
- 2 find offset values and promote to StVEC insts when possible

# StVEC Code Generation

**Input:** basic intrinsics loop

## The proposed algorithm

- 1 replace every unaligned reference by two aligned loads
- 2 find offset values and promote to StVEC insts when possible

Some additional optimizations:

- dead-code elimination
- 3-stage software-pipelining

# StVEC Code Generation

**Input:** basic intrinsics loop

## The proposed algorithm

- 1 replace every unaligned reference by two aligned loads
- 2 find offset values and promote to StVEC insts when possible

Some additional optimizations:

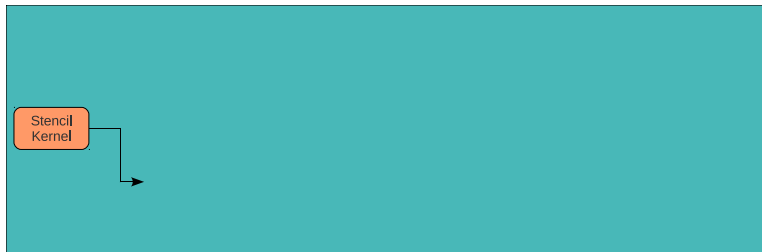
- dead-code elimination
- 3-stage software-pipelining

## Properties

- can be emulated w/ existing vector ISA
- unaligned loads can be eliminated

# Evaluation Methodology

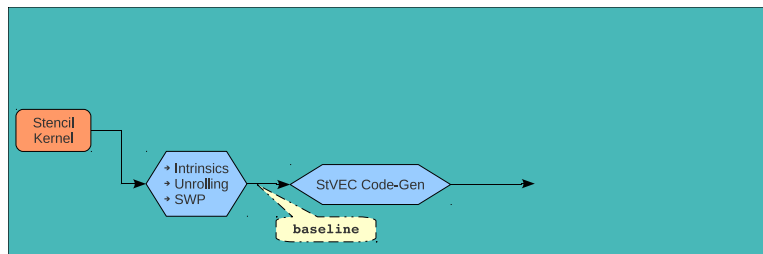
# Emulating StVEC Instructions on Real Vector ISA



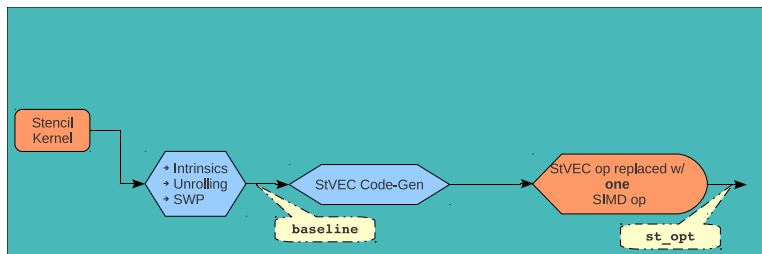
# Emulating StVEC Instructions on Real Vector ISA



# Emulating StVEC Instructions on Real Vector ISA

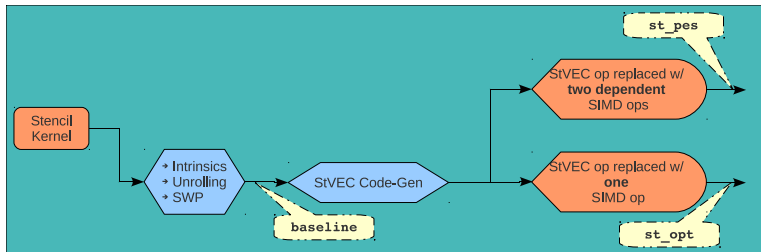


# Emulating StVEC Instructions on Real Vector ISA



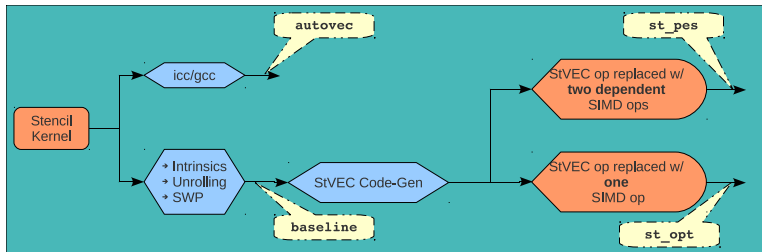
- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt:** StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$

# Emulating StVEC Instructions on Real Vector ISA



- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt:** StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
- **st-pes:** StVEC inst models delay of 2 dependent SIMD insts
  - $T_{StVRF} \leq 2 * T_{cycle}$

# Emulating StVEC Instructions on Real Vector ISA



- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt:** StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
- **st-pes:** StVEC inst models delay of 2 dependent SIMD insts
  - $T_{StVRF} \leq 2 * T_{cycle}$

# Setup

## Running on x86 architectures

- Intel Core i7 Nehalem, Intel Sandy Bridge, Intel Core2 Quad
- AMD Phenom (K10)

## Stencil Benchmarks

- 1D: Jacobi (2-, 3-, 5- and 7-point)
- 2D: Jacobi (5- and 9-point), POP, FDTD 2D, Rician Denoise 2D
- 3D: Jacobi (27-point), Heattut 3D

## L1-resident problem size

- assume tiling was performed beforehand if necessary

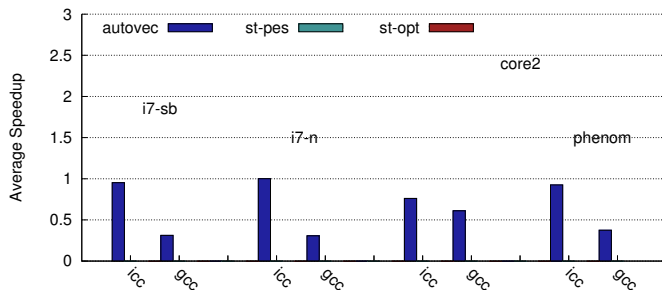
## Compilers

- ICC 12 (w/ *-fast*) and GCC 4.4.4 (w/ *-O3*)

## Experimental Results

# Average (Geometric Mean) Speedup with StVEC

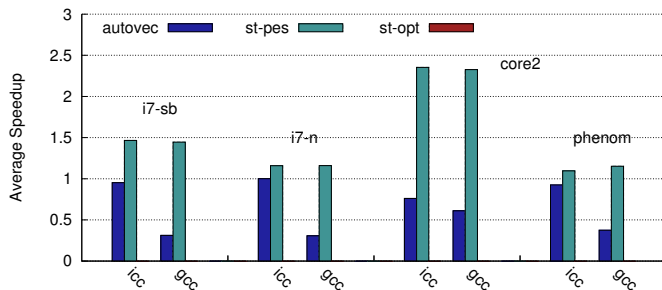
Single-precision (average across all 12 benchmarks)



- Normalized to **baseline** (intrinsic + unrolling + SWP)

# Average (Geometric Mean) Speedup with StVEC

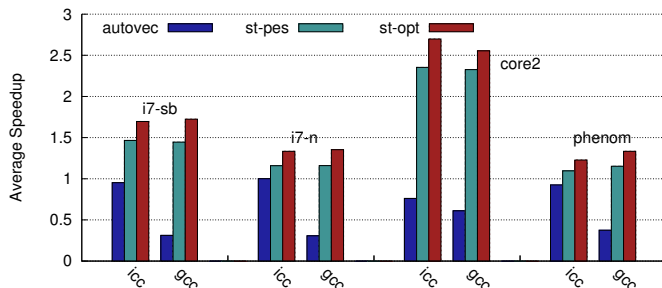
Single-precision (average across all 12 benchmarks)



- Normalized to **baseline** (intrinsic + unrolling + SWP)

# Average (Geometric Mean) Speedup with StVEC

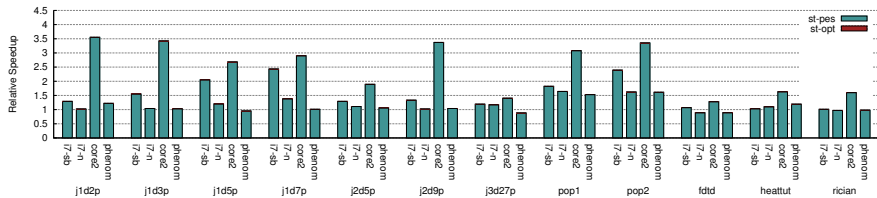
Single-precision (average across all 12 benchmarks)



- Normalized to **baseline** (intrinsic + unrolling + SWP)
- Single-precision: 7% to 2.26x for st-pes and 20% to 2.47x for st-opt
- Double-precision: 30% to 65% for st-opt

# Speedup with ICC Across Machines

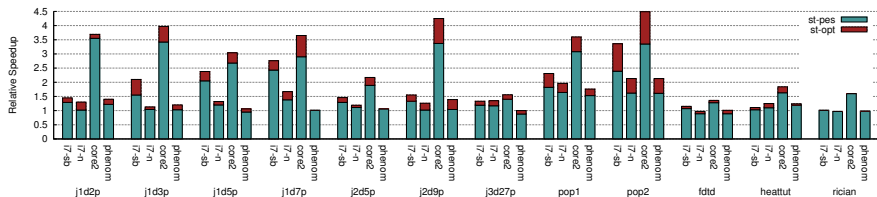
## Single-precision



- Normalized to **baseline** (intrinsic + unrolling + SWP)

# Speedup with ICC Across Machines

## Single-precision



- Normalized to **baseline** (intrinsic + unrolling + SWP)

# StVEC Hardware Overhead

## StVRF access time in 45nm CMOS

- $T_{B\text{VRF}}$ : SRAM model in CACTI
- $T_{\text{VRA}}$ : circuit synthesis/layout by Synopsys Design Compiler

| #Regs.       | #Banks | Reg. size | BVRF    | StVRF   |
|--------------|--------|-----------|---------|---------|
| 128 (st-opt) | 4      | 128-bit   | 0.24 ns | 0.30 ns |
| 256 (st-pes) | 8      | 256-bit   | 0.37 ns | 0.50 ns |

- $T_{\text{StVRF}} \approx T_{\text{BVRF}} + T_{\text{VRA}}$
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - $T_{\text{StVRF}} \leq T_{\text{cycle}}$
- **st-pes**: StVEC inst models delay of 2 dependent SIMD inst
  - $T_{\text{StVRF}} \leq 2 * T_{\text{cycle}}$

# StVEC Hardware Overhead

## StVRF access time in 45nm CMOS

- $T_{BVRF}$ : SRAM model in CACTI
- $T_{VRA}$ : circuit synthesis/layout by Synopsys Design Compiler

| #Regs.       | #Banks | Reg. size | BVRF    | StVRF   |
|--------------|--------|-----------|---------|---------|
| 128 (st-opt) | 4      | 128-bit   | 0.24 ns | 0.30 ns |
| 256 (st-pes) | 8      | 256-bit   | 0.37 ns | 0.50 ns |

- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
  - Ex1:  $F = 3\text{GHz}$ ,  $T_{cycle} = 0.33\text{ns} > T_{StVRF} \Rightarrow$  **No overhead!**
- **st-pes**: StVEC inst models delay of 2 dependent SIMD inst
  - $T_{StVRF} \leq 2 * T_{cycle}$

# StVEC Hardware Overhead

## StVRF access time in 45nm CMOS

- $T_{BVRF}$ : SRAM model in CACTI
- $T_{VRA}$ : circuit synthesis/layout by Synopsys Design Compiler

| #Regs.       | #Banks | Reg. size | BVRF    | StVRF   |
|--------------|--------|-----------|---------|---------|
| 128 (st-opt) | 4      | 128-bit   | 0.24 ns | 0.30 ns |
| 256 (st-pes) | 8      | 256-bit   | 0.37 ns | 0.50 ns |

- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
  - Ex1:  $F = 3\text{GHz}$ ,  $T_{cycle} = 0.33\text{ns} > T_{StVRF} \Rightarrow$  **No overhead!**
- **st-pes**: StVEC inst models delay of 2 dependent SIMD inst
  - $T_{StVRF} \leq 2 * T_{cycle}$
  - Ex2:  $F = 3\text{GHz}$ ,  $T_{cycle} = 0.33\text{ns} < T_{StVRF} \Rightarrow$  **Extra cycle overhead!**

# StVEC Hardware Overhead

## StVRF access time in 45nm CMOS

- $T_{BVRF}$ : SRAM model in CACTI
- $T_{VRA}$ : circuit synthesis/layout by Synopsys Design Compiler

| #Regs.       | #Banks | Reg. size | BVRF    | StVRF   |
|--------------|--------|-----------|---------|---------|
| 128 (st-opt) | 4      | 128-bit   | 0.24 ns | 0.30 ns |
| 256 (st-pes) | 8      | 256-bit   | 0.37 ns | 0.50 ns |

- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
  - Ex1:  $F = 3\text{GHz}$ ,  $T_{cycle} = 0.33\text{ns} > T_{StVRF} \Rightarrow$  **No overhead!**
- **st-pes**: StVEC inst models delay of 2 dependent SIMD inst
  - $T_{StVRF} \leq 2 * T_{cycle}$
  - Ex2:  $F = 3\text{GHz}$ ,  $T_{cycle} = 0.33\text{ns} < T_{StVRF} \Rightarrow$  **Extra cycle overhead!**
  - Ex3:  $F = 2\text{GHz}$ ,  $T_{cycle} = 0.50\text{ns} \geq T_{StVRF} \Rightarrow$  **No overherad!**

# Conclusion

## Take-home Message

- Vectorization of stencils is expensive
  - Previous solutions: unaligned loads or shuffle instructions!
- **Our solution: StVEC** – new vector instruction extension
  - Fast execution of stencils
  - Small hardware changes
  - Eliminating unaligned loads

# Conclusion

## Take-home Message

- Vectorization of stencils is expensive
  - Previous solutions: unaligned loads or shuffle instructions!
- **Our solution: StVEC** – new vector instruction extension
  - Fast execution of stencils
  - Small hardware changes
  - Eliminating unaligned loads
- Performance evaluation with existing x86 vector ISAs
  - Optimistic (1 StVEC inst  $\approx$  1 SIMD inst): 20% to 2.47x
  - Pessimistic (1 StVEC  $\approx$  2 dependent SIMD insts): 7% to 2.26x
- Best fit for 128-bit wide vector computations
  - May require additional pipeline stage(s) for wider vectors

Questions?